Chapter 8

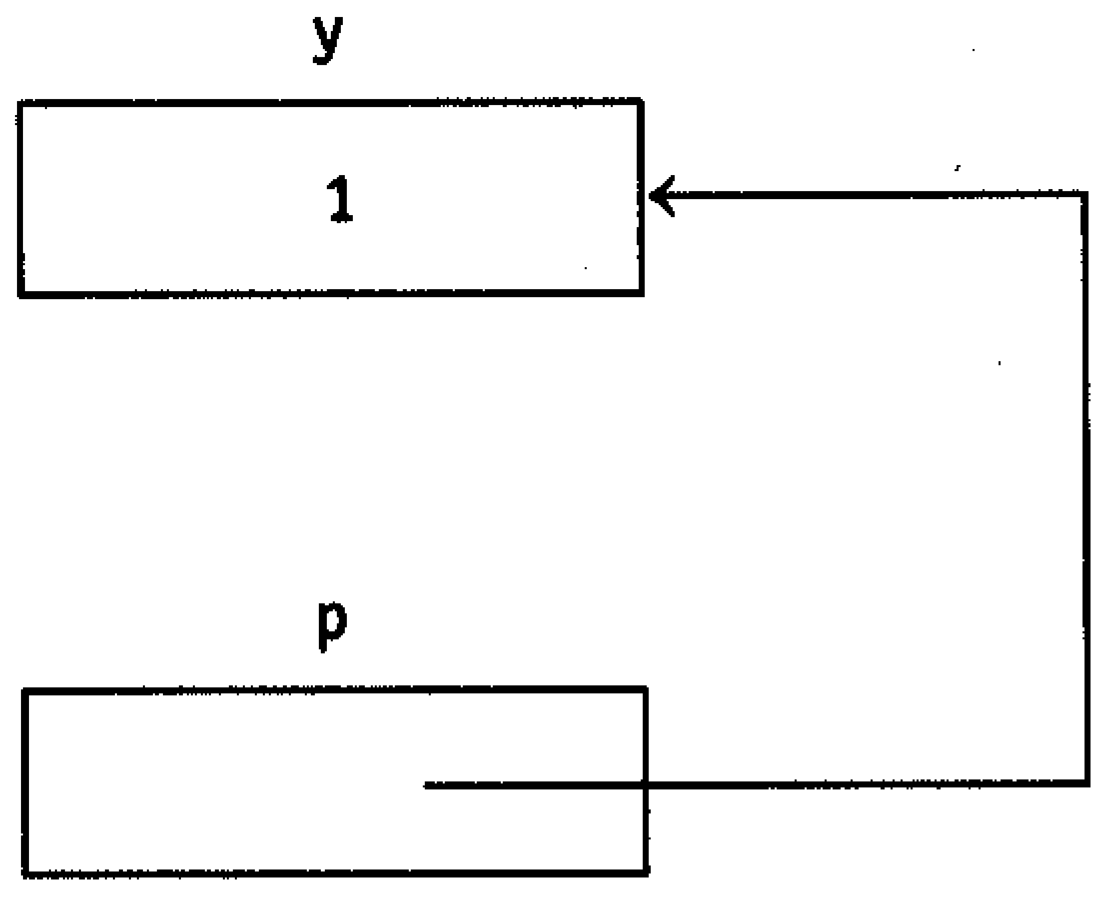
Evaluating the Instruction Set Architecture of H1: Part 2

An argument in call by value can be &y. For example,

```
int y = 1;
void fr(int *p) {
    *p = *p + 5;        // accesses y
 }
void main() {
        fr(&y);
}
```

Parameter p points to y on entry into fr. y can be accessed by dereferencing p.

FIGURE 8.2



The special case of passing the address of an argument is referred to as *call by address* or *call by reference.*

The program on the next slide contains both call by value and call by reference. fv uses call by value; fr uses call by reference; fvr uses both call by value and call by reference.

FIGURE 8.1

```
#include <iostream>
using namespace std;

int y = 1;
void fv(int x)
{
      x = x + 5;
}
void fr(int *p)
{
      *p = *p + 5;        // dereference pointer to access y
}
void fvr(int x, int *p)
{
      *p = x;
}
void main()
{
      fv(y);                    // call by value-pass value
      cout << "y = " << y << endl;
      fr(&y);                   // call by reference-pass address
      cout << "y = " << y << endl;
      fvr(20, &y);              // call by value and reference
      cout << "y = " << y << endl;
}
```

Let's examine the assembly language needed for the `fv` and `fr` functions. To call `fv` from `main`, we need

```
ld y        ; get value of y
push        ; create and initialize parameter x
call fv
dloc 1      ; remove x
```

To call `fr`, we need

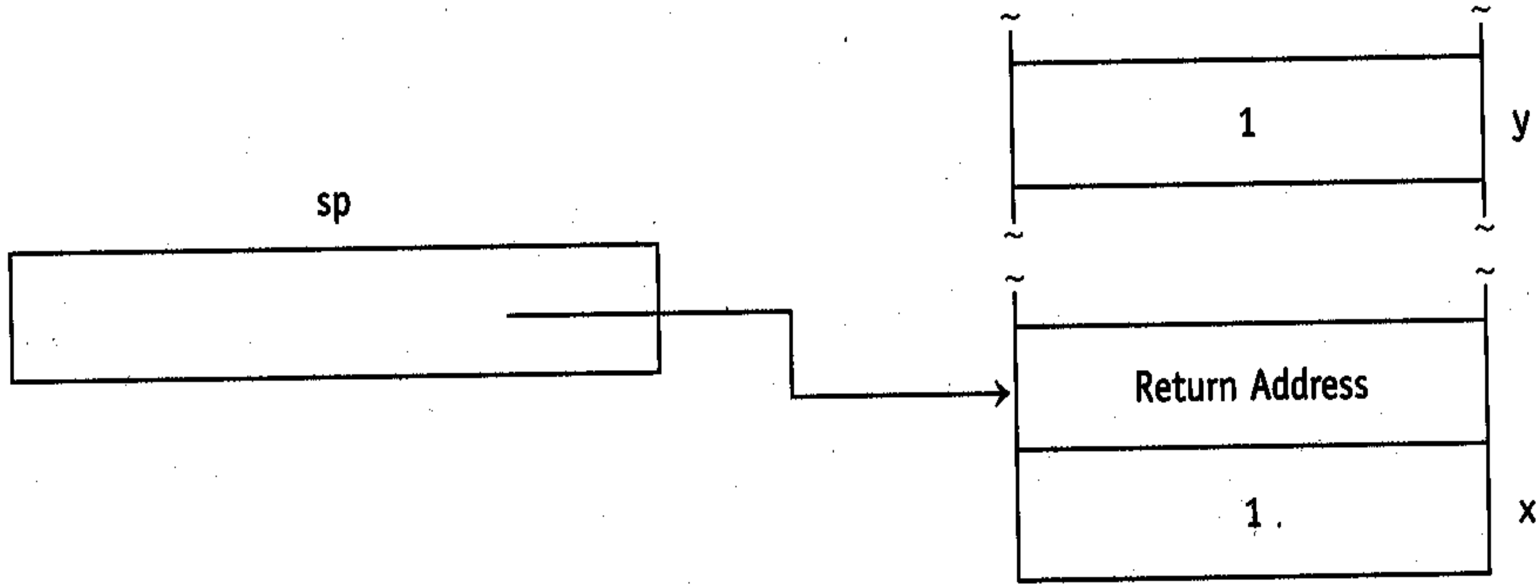
```
ldc y       ; get the address of y
push        ; create and initialize parameter p
call fr
dloc 1      ; remove p
```

On entry into fv, the relative address of x is 1.
x contains the *value* of y.

On entry into fr, the relative address of p is 1.
p *points to* y.

See the next slide.

FIGURE 8.3 a) On entry into fv



b) On entry into fr

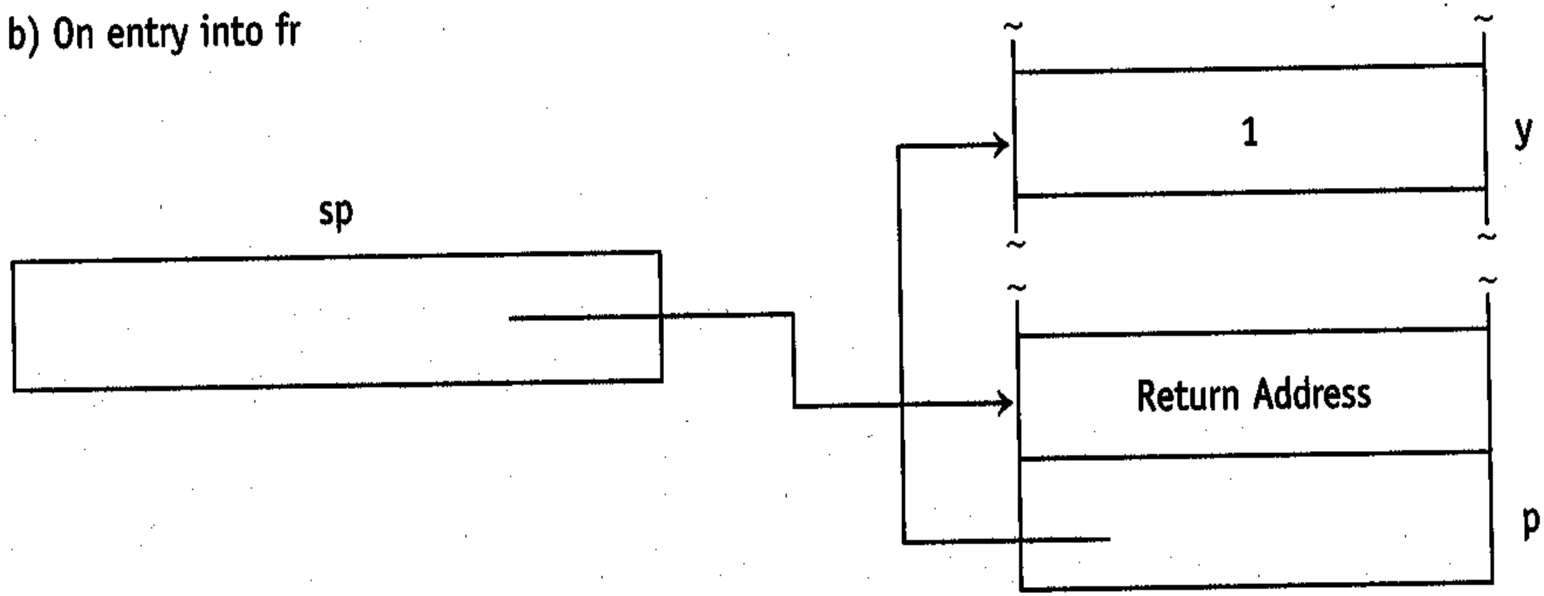


FIGURE 8.5

```
                                ; x = x + 5;
fv:          ldc   5            ; get 5
             addr  1            ; add x
             str   1            ; store in x

             ret
;==============================================================
                                ; *p = *p + 5;
fr:          ldr   1            ; get p
             ldi                ; get *p
             add   @5           ; add 5
             push               ; prepare for sti
             ldr   2            ; get p
             sti                ; store into *p

             ret
;==============================================================
                                ; *p = x;
fvr:         ldr   1            ; get x
             push               ; prepare for sti
             ldr   3            ; get p
             sti                ; store into *p

             ret
;==============================================================
main:        ld    y            ; fv(y);
             push
             call  fv
             dloc  1

             ldc   @m0          ; cout << "y = " << y << endl;
             sout
             ld    y
             dout
             ldc   '\n'
             aout

             ldc   y            ; fr(&y);
             push
             call  fr
             dloc  1

             ldc   @m1          ; cout << "y = " << y << endl;
```

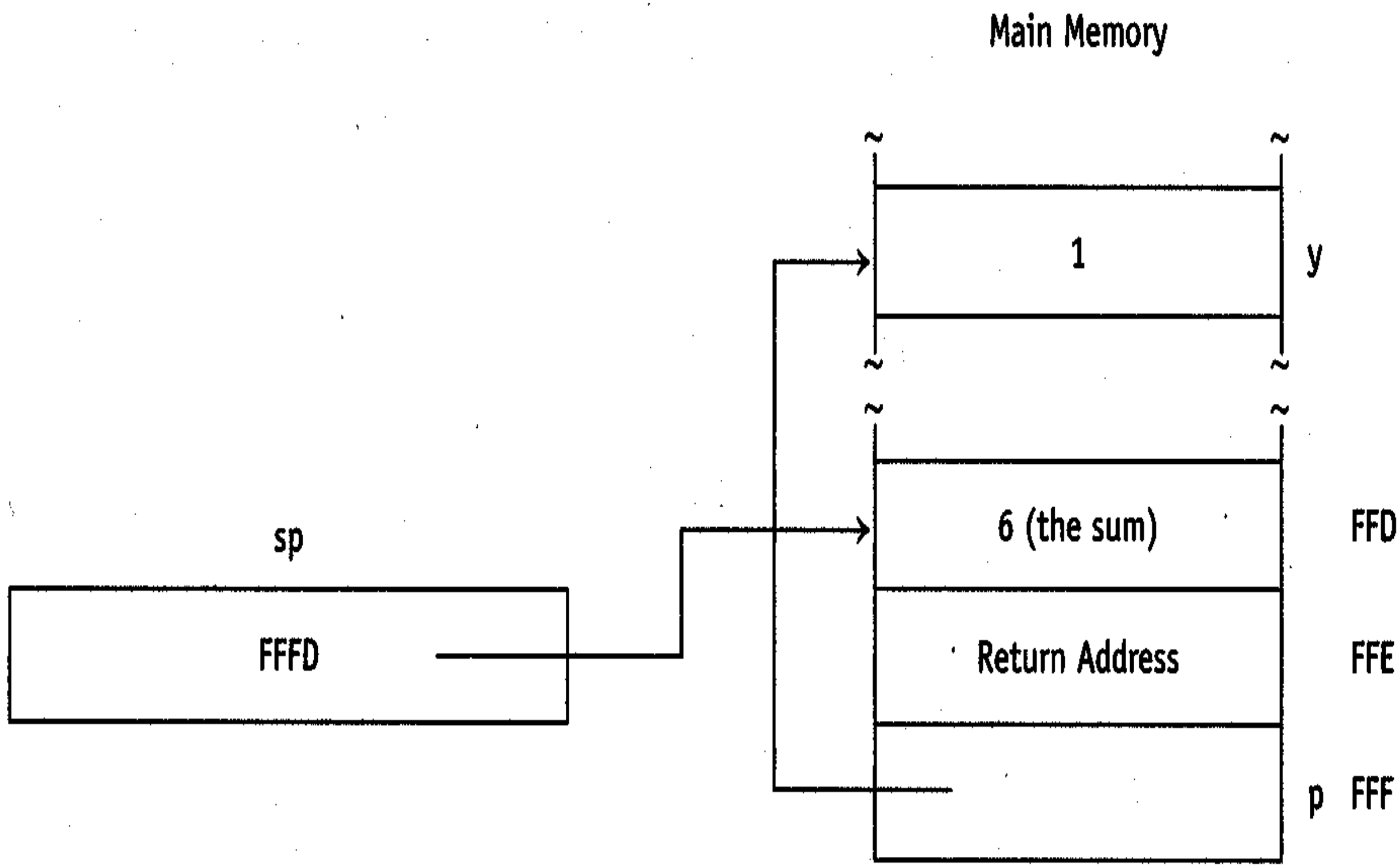
(continued)

FIGURE 8.5 (continued)

```
        sout
        ld    y
        dout
        ldc   '\n'
        aout
        ldc   y            ; fvr(20, &y);
        push
        ldc   20
        push
        call  fvr
        dloc  2
        ldc   @m2          ; cout << "y = " << endl;
        sout
        ld    y
        dout
        ldc   '\n'
        aout
        halt
@5:     dw    5
y:      dw    1
@m0:    dw    "y = "
@m1:    dw    "y = "
@m2:    dw    "y = "
        end   main
```

Relative address of p changes from 1 to 2 during the execution of fr because of the push.

FIGURE 8.4



Reference parameters

- Receives *address* of corresponding argument.
- Preceded by ‘&’ in the parameter list.
- Whenever a reference parameter is used in the called function, the address it contains is dereferenced.

The address of y is passed because z is a reference parameter.

```
void fr (int &z)
{
   z = z + 5;
 }

void main()
{
   fr(y);          // address of y is passed.
}
```

z is dereferenced wherever it is used because z is a reference parameter.

```
void fr(int  &z)
{
   z = z + 5;
}
```

FIGURE 8.6

```
#include <iostream>
using namespace std;

int y = 1;
void fv(int x)                          // x is a value parameter
{
      x = x + 5;
}
void fr(int &z)                         // z is a reference parameter
{
      z = z + 5;
}
void fvr(int x, int &z)                 // x is value, z is ref parameter
{
      z = x;
}
void main()
{
      fv(y);                            // call by value
      cout << "y = " << y << endl;      // y is still 1
      fr(y);                            // call by reference
      cout << "y = " << y << endl;      // y is now 6
      fvr(20, y);
      cout << "y = " << y << endl;      // y is now 20
}
```

Conceptual view of reference parameter mechanism

- The argument itself is passed, replacing the corresponding parameter wherever it appears in the called function.
- This is not what really happens, but it is easy to understand.

Line 7 becomes x = x + 5; on the 1st call of fr.
It becomes y = y + 5; on the second call.

FIGURE 8.7

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 int x = 5, y = 6;
 5 void fr(int &z)  // z is a reference parameter
 6 {
 7       z = z + 5;
 8 }
 9 void main()
10 {
11       fr(x);                          // x is the argument
12       fr(y);                          // y is the argument
13 }
```

An exception to the rule that whenever a reference parameter is used, the address it contains is dereferenced is illustrated by the program in the next slide.

y is not dereferenced on line 12 because it is also the argument corresponding to the reference parameter z on line 5.

FIGURE 8.8

```
#include <iostream>
using namespace std;

int x = 100;
void fr(int &z)
{
      z = z + 5;          // adds 5 to x
}
void ft(int &y)
{
      y = y + 1;          // adds 1 to x
      fr(y);              // reference parameter now is argument
}
void main()
{
      ft(x);
      cout << "x = " << x << endl;
}
```

Call by value vs Call by reference

- Call by value is a “one-way street”.
- Call by reference is a “two-way street”.
- Call by value duplicates the argument. It is time and space *inefficient* for large arguments.
- Call by reference passes only the address of the argument.
- Call by reference requires time-consuming dereferencing operations.

How does call by reference handle the following calls?

fr2(x);

fr2(200);

fr2(x + y);

For the 2nd and 3rd calls, the value of the argument is placed on the stack, and then the address of this value is passed.

FIGURE 8.9

```
#include <iostream>
using namespace std;

int x = 1, y = 2;
void fv2(int z)        // uses call by value
{
     cout << "z = " << z << endl;
     z = 5;
}
void fr2(int &z)       // uses call by reference
{
     cout << "z = " << z << endl;
     z = 5;
}
void main()
{
     fv2(x);           // z = 1   is displayed
     fv2(100);         // z = 100 is displayed
     fv2(x + y);       // z = 3   is displayed
     fr2(x);           // z = 1   is displayed, x is assigned 5
     fr2(100);         // z = 100 is displayed
     fr2(x + y);       // z = 7   is displayed
}
```

FIGURE 8.10

```
fv2:        ldc  @m0       ; cout << "z = " << z << endl;
            sout
            ldr  1
            dout
            ldc  '\n'
            aout

            ldc  5              ; z = 5;
            str  1

            ret
;==========================================================
                           ; cout << "z = " << z << endl;
fr2:        ldc  @m1            ; get address of @m1
            sout                ; display "z ="
            ldr  1              ; get z
            ldi                 ; dereference z
            dout                ; display it
            ldc  '\n'           ; newline
            aout

```

(continued)

FIGURE 8.10 (continued)

```
          ldc   5          ; z = 5;
          push
          ldr   2
          sti

          ret
;==================================================================
main:     ld    x          ; fv2(x);
          push
          call  fv2
          dloc  1

          ldc   100        ; fv2(100);
          push
          call  fv2
          dloc  1

          ld    x          ; fv2(x+y);
          add   y
          push
          call  fv2
          dloc  1

          ldc   x          ; fr2(x);
          push
          call  fr2
          dloc  1

                           ; fr2(100);
          ldc   100                ; get 100
          push                     ; create and init implicit var on stack
          swap                     ; get sp
          st    @spsave            ; save it
          swap                     ; restore sp
          ld    @spsave            ; get address of implicit var on tos
          push                     ; pass this address to fr2
          call  fr2
          dloc  2                  ; deallocate parameter and implicit variable

                                   ; fr2(x+y);
          ld    x                  ; get x
          add   y                  ; add y
```

(continued)

FIGURE 8.10 (continued)

```
          push                 ; create and init implicit var on stack
          swap                 ; get sp
          st    @spsave        ; save it
          swap                 ; restore sp
          ld    @spsave        ; get address of implicit var on tos
          push                 ; pass this address to fr2
          call fr2
          dloc 2               ; deallocate parameter and implicit variable

          halt
x:        dw    1
y:        dw    2
@m0:      dw    "z = "
@m1:      dw    "z = "
@spsave:  dw    0
          end   main
```

Function overloading

The use of the same name for more than one function. The functions with a common name must have parameter lists than differ in type, order, and/or number.

FIGURE 8.11

```
#include <iostream>
using namespace std;

int x = 1;
void fol()                      // fol with no parameters
{
      x = x + 1;
}
void fol(int n)                 // fol with one int parameter
{
      x = x + n;
}
void fol(int n, int m)          // fol with two int parameters
{
      x = x + n + m;
}
void main()
{
      fol(10);                  // calls 2nd fol function
      cout << x << endl;
      fol();                    // calls 1st fol function
      cout << x << endl;
      fol(2, 3);                // calls 3rd fol function
      cout << x << endl;
}
```

Implementing function overloading

- Function overloading is implemented by means of *name mangling.*
- With name mangling, each function name at the assembly level has a mangled name. The mangled name includes an encoding of the parameter list.
- Thus, functions with the same name but different parameter lists will have distinct names at the assembly level.

The assembly-level names for

```
void fol() {…}
void fol(int n) {…}
void fol(int n, int m) {...}
void fol(int *p) {...}
void fol(int &z) {...}
void fol(float q) {...}
```

are

```
@fol$v
@fol$i
@fol$ii
@fol$pi
@fol$ri
@fol$f
```

FIGURE 8.12

Type	**Encoding**
void	v
boolean	b
signed char	zc
unsigned char	uc
char	compiler dependent: zc or uc depending on how compiler treats **char** (see Section 1.9)
int	i
signed int	i
unsigned int	ui
short int	s
signed short	s
signed short int	s
unsigned short	us
unsigned short int	us
long	l
long int	l
signed long int	l
unsigned long	ul
unsigned long int	ul
float	f
double	d
long double	g
pointer	prefix p
reference	prefix r

FIGURE 8.13

```
@fol$v:     ldc  1              ; x = x + 1;
            add  x
            st   x

            ret
;=========================================================
@fol$i:     ld   x              ; x = x + n;
            addr 1
            st   x

            ret
;=========================================================
@fol$ii:    ld   x              ; x = x + n + m;
            addr 1
            addr 2
            st   x

            ret
;=========================================================
main:       ldc  10             ; fol(10);
            push
            call @fol$i
            dloc 1

            ld   x              ; cout << x << endl;
            dout
            ldc  '\n'
            aout
```

(continued)

FIGURE 8.13 (continued)

```
        call @fol$v         ; fol();

        ld   x              ; cout << x << endl;
        dout
        ldc  '\n'
        aout

        ldc  3              ; fol(2, 3);
        push
        ldc  2
        push
        call @fol$ii
        dloc 2

        ld   x              ; cout << x << endl;
        dout
        ldc  '\n'
        aout

        halt
x:      dw   1
        end  main
```

C++ struct

- A user-defined type consisting of fields, each of which with its own type.
- The dot operator (‘.’)is used to access a field of a struct variable, given its name.
- The pointer operator (‘->’) is used to access a field of a struct variable, given its address.
- Successive fields are mapped to successive locations.

FIGURE 8.14

```
struct Coordinates {
        int x;
        int y;
};
Coordinates gs;                           // global struct
void tests(Coordinates *ps)
{
        Coordinates ls;                   // local struct
        int li = 5;                       // local int
        ls.y = 4;
        ps -> y = li;
}
void main()
{
        tests(&gs);
        gs.y = 3;
}
```

Mangled name for the tests function in the preceding slide is

@tests$p11Coordinates

```
struct iii {
…
};
void f(iii x) {
…
}
```

What’s the mangled name for f?

@f$iii ← 3 int’s or struct iii?

or

@f$3iii ← not ambiguous

FIGURE 8.15

```
@tests$p11Coordinates:
            aloc 2          ; Coordinates ls

            ldc  5          ; int li = 5;
            push

                            ; ls.y = 4;
            ldc  4              ; get [redacted] 4
            str  2              ; store into ls.y

                            ; ps -> y = li;
            ldr  0              ; get li
            push                ; push it
            ldc  1              ; get offset of y field
            addr 5              ; add ps
            sti                 ; assign li to ps -> y

            dloc 3              ; deallocate locals
            ret
;=================================================================
main:       ldc  gs          ; tests(&gs);
            push
            call @tests$p11Coordinates
            dloc 1

            ldc  3           ; gs.y = 3
            st   gs + 1

            halt
gs:         dw   2 dup 0
            end  main
```

A struct can be returned with a return statement in C++. Requires that the address of a return area be passed to the called function.

See the program on the next slide.

FIGURE 8.16

```
struct S {
    int x;
    int y;
};
S t;
S ret_struct() {
    S s;
    s.x = 1;
    s.y = 2;
    return s;      // return a struct
}
void main()
{
    t = ret_struct();
}
```

FIGURE 8.17

```
@ret_struct$v:
            aloc 2       ; S s

            ldc  1       ; s.x = 1;
            str  0

            ldc  2       ; s.y = 2;
            str  1

                         ; return s;
            ldr  1           ; get s.y
            push             ; push s.y
            ldr  1           ; get s.x
            push             ; push s.x
            ldr  5           ; get address of return area
            sti              ; pop value of s.x into return area
            add  @1          ; get address of next word in return area
            sti              ; pop value of s.y into return area
            dloc 2           ; deallocate s
            ret
;================================================================
main:

                          ; t = ret_struct();
            ldc  t            ; get address of t
            push              ; create implicit parameter
            call @ret_struct$v
            dloc 1            ; deallocate parameter

            halt
t:          dw         2 dup 0
@1:         dw         1
@spsave:    dw         0
            end        main
```

Are there pointers in Java?

Reference parameters in C++ hide the passing and dereferencing of an address. Java hides its pointers in a similar fashion.

On the next slide, both the C++ and Java programs dereference an address. In the C++ program, this process is explicit. In the Java program, this process is hidden.

FIGURE 8.18 a) C+ program with pointer p

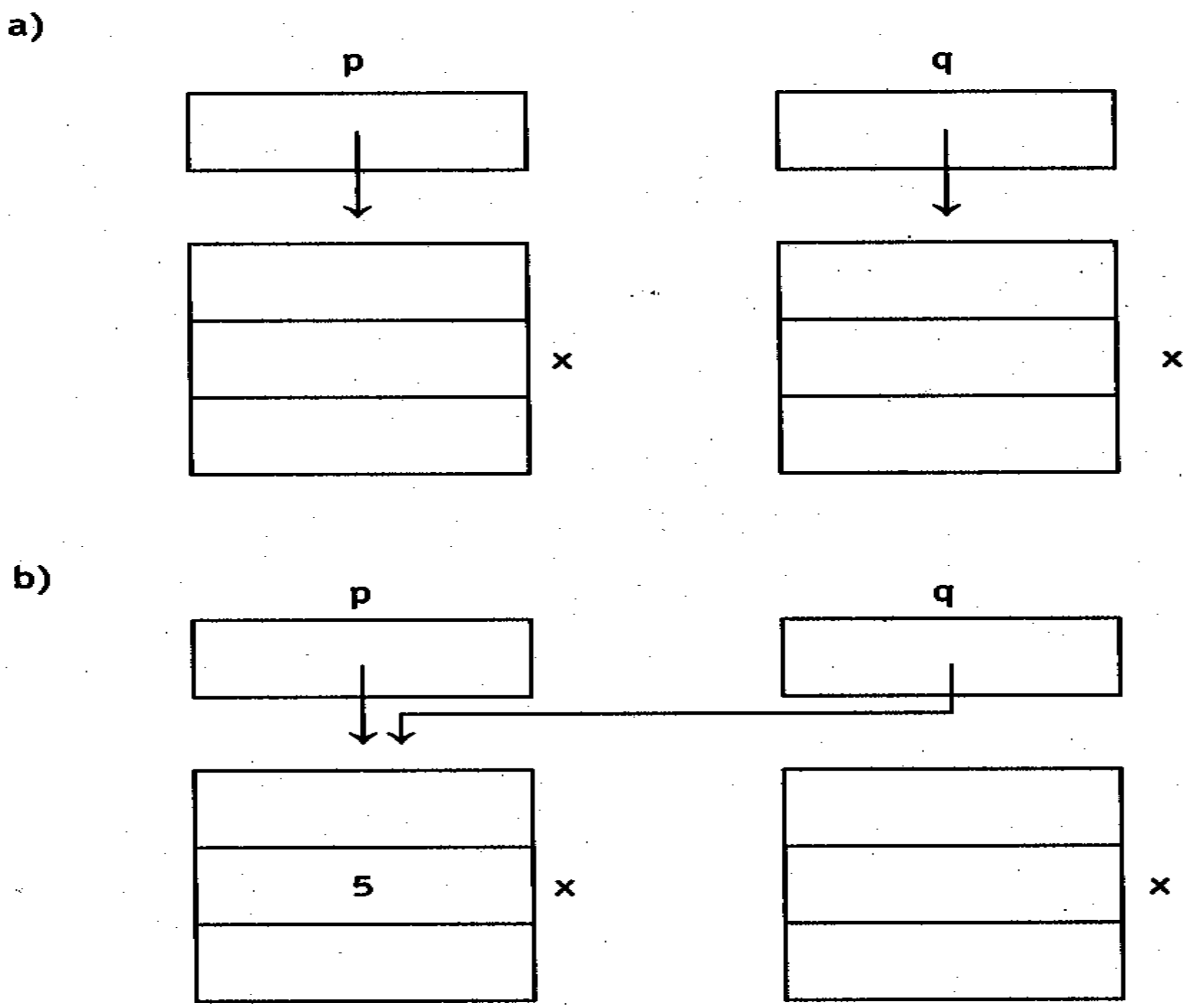
```
class S {
 public:
 int x;
 int y;
};
void main()
{
 S *p;              // declare a pointer p
 p = new S();       // create object and assign its address to p
 p -> x = 3;        // (*p).x = 3;
 p -> y = 4;        // (*p).y = 4;
}
```

b) Java program with reference p

```
class S {
 int x;
 int y;
}

class Pex {
   public static void main(String arg[]) {
      S p;                    // create reference p
      p = new S();            // assign p an instance of S
      p.x = 3;                // access x via p
      p.x = 4;                // access y via p
   }
}
```

p and q are Java references. What happens when
q = p; is executed?

FIGURE 8.19



Pointers to functions

- Allowed in C++
- The C++ compiler translates the name of a function not followed by parenthesis to a pointer to that function (i.e., to its address). For example, **fp(3);** is a function call, but **fp** is a pointer to the fp function.

Declaring a pointer to a function

```
int (*p)(int x);
```

p is a pointer to a function that is passed an int and returns an int.

FIGURE 8.20

```
#include <iostream>
using namespace std;

int y;
int (*p)(int);
int fp(int x)
{
    cout << x << endl;
    return x;
}
void main()
{
    y = fp(3);      // call fp
    p = fp;
    y = p(3);       // call fp via p pointer
}
```

y = p(3);

is difficult to translate, where p is a pointer to a function.

```
ld      @call
add     p            ;  the required call inst now in ac
st      * + 3
ldc     3
push
dw      0            ; changes to a call
dloc    1
st      y
```

where

```
@call:    dw    call 0
```

Pointer arithmetic in C++

Arithmetic on pointers in C++ uses the size of the pointed to type as the unit size. For example, if p is a pointer to a struct consisting of 10 words, then

```
p = p + 1;
```

adds 10 (not 1) to p.

An array name is an address

The name of an array without the square brackets points to the 1st slot of the array. For example, given

```
int table[3];
```

Then table by itself is the address of table[0]. That is,

table (type is int pointer)

and

&table[0] (type is int pointer)

are equivalent.

Using array name as a pointer

The name of an array can be used as a pointer to that array (because it really is a pointer to the array). For example, given int table[3]; then

```
*(table+ 2) = 99;
```

and

```
table[2] = 99;
```

are equivalent.

Using a pointer to an array as if it were the name of the array

A pointer to an array can be used as if it were the name of an array.

```
int *p;
int   table[3];
p = table;            /* p now points to table
```

p and table have the same value and same type

so we can use them interchangeably (almost). */

```
p[2] = 99;  // use p as if it were an array name.
```

```
int table[3];
int *p;
p = table;
int x;
```

p is a *pointer variable*; table is a *pointer constant*.
So table cannot be assigned a new value.

```
p = table + 1;   // legal because p is a variable
table = &x;      // illegal, table always points to table[0]
```

Creating a global array (static local array is similar)

```
int table[3];
table[2] = 7;
```

is translated to

```
ldc  7
st   table + 2

table: dw    3 dup 0
```

Creating a dynamic local array

```
void f() {
   int table[3];
   table[2] = 7;
}
```

is translated to

```
   aloc  3
   ldc 7
   str  2
```

Accessing dynamic local array

FIGURE 8.21

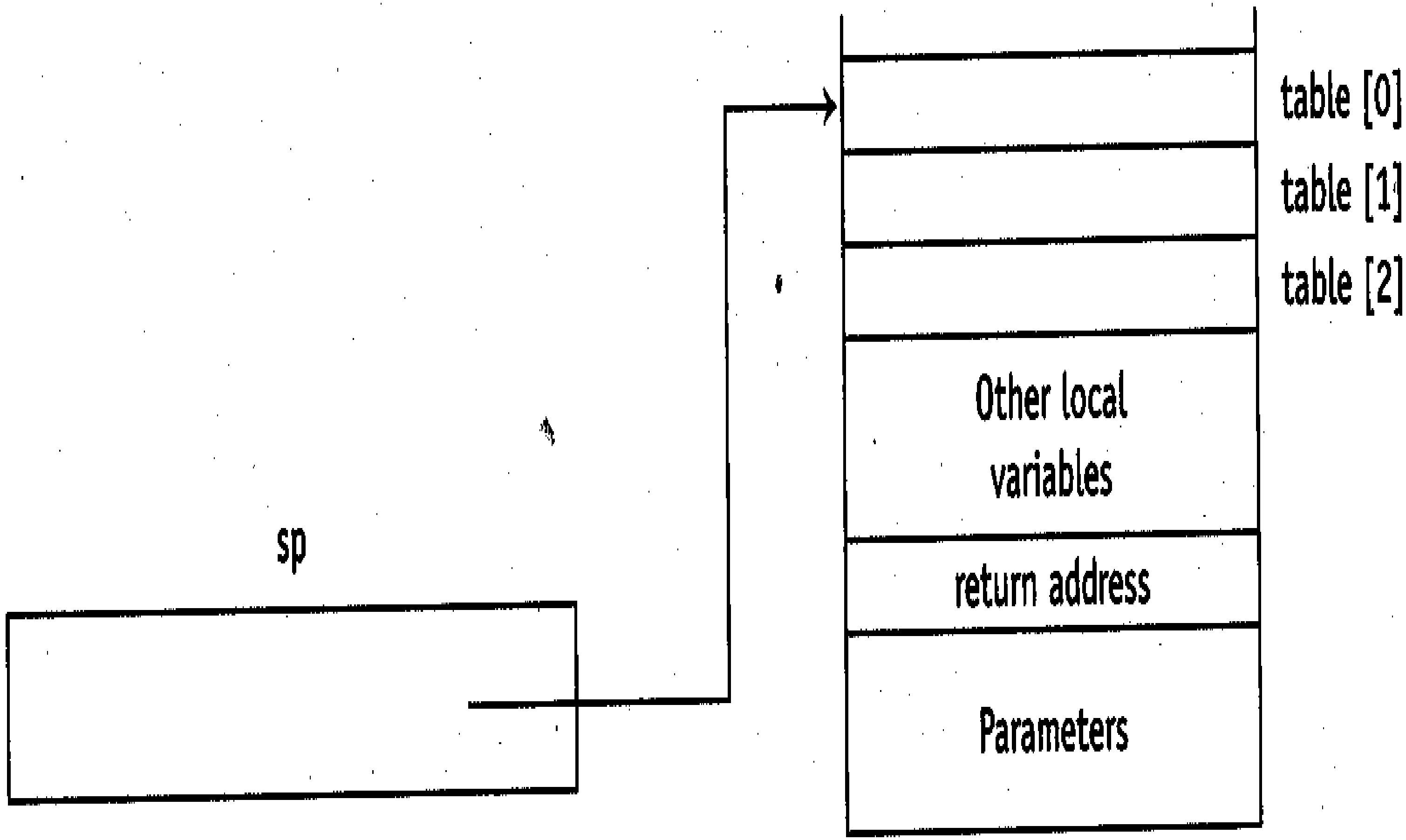


FIGURE 8.22

```
#include <iostream>
using namespace std;

int ga[3], x;                          // ga is global array
void arrays()
{
      static int sla[3];               // sla is a static local array
      int dla[3];                      // dla is a dyn local array

      cout << "enter index\n";
      cin >> x;

      ga[2] = 99;
      ga[x] = 99;

      sla[2] = 99;
      sla[x] = 99;

      dla[2] = 99;
      dla[x] = 99;

}
void main ()
{
   arrays();
}
```

Must compute the address of
ga[x]
at run time because the value of
x is not known until run time.

FIGURE 8.23

```
@arrays$v:
            aloc 3              ; int [redacted] dla[3];

            ldc  @m0            ; cout << "enter index\n";
            sout

            din                 ; cin >> x;
            st   x

            ldc  99             ; ga[2] = 99;
            st   ga + 2

            ldc  99             ; ga[x] = 99;
            push
            ldc  ga
            add  x
```

(continued)

FIGURE 8.23
(continued)

```
                sti

                ldc   99              ; sla[2] = 99;
                st    @s0_sla + 2

                ldc   99              ; sla[x] = 99;
                push
                ldc   @s0_sla
                add   x
                sti

                ldc   99              ; dla[2] = 99;
                str   2

                                      ; dla[x] = 99;
                ldc   99                ; get 99
                push                    ; prepare for sti
                swap                    ; get sp
                st    @spsave           ; save it
                swap                    ; restore sp
                ldc   1                 ; get rel address of dla[0]
                add   @spsave           ; get abs address of dla[0]
                add   x                 ; get abs address of dla[x]
                sti                     ; store 99 into dla[x]

                dloc 3
                ret
;===========================================================================
main:           call @arrays$v        ; arrays();

                halt
ga:             dw    3 dup 0         ; global array
          sla
@s0_[redacted]: dw    3 dup 0         ; static local array
x:              dw    0
@m0:            dw    "enter index\n"
@spsave:        dw    0
                end   main
```

Passing an array—really passing a pointer to 1st slot

```
int table[3];
…
  f(table);
```

Passing a pointer to the first slot of table. Corresponding argument should be an int pointer.

Empty square brackets in an array parameter are interpreted by the compiler as meaning pointer

```
void f(int t[ ]) {          // t is an int *
  …
}
```

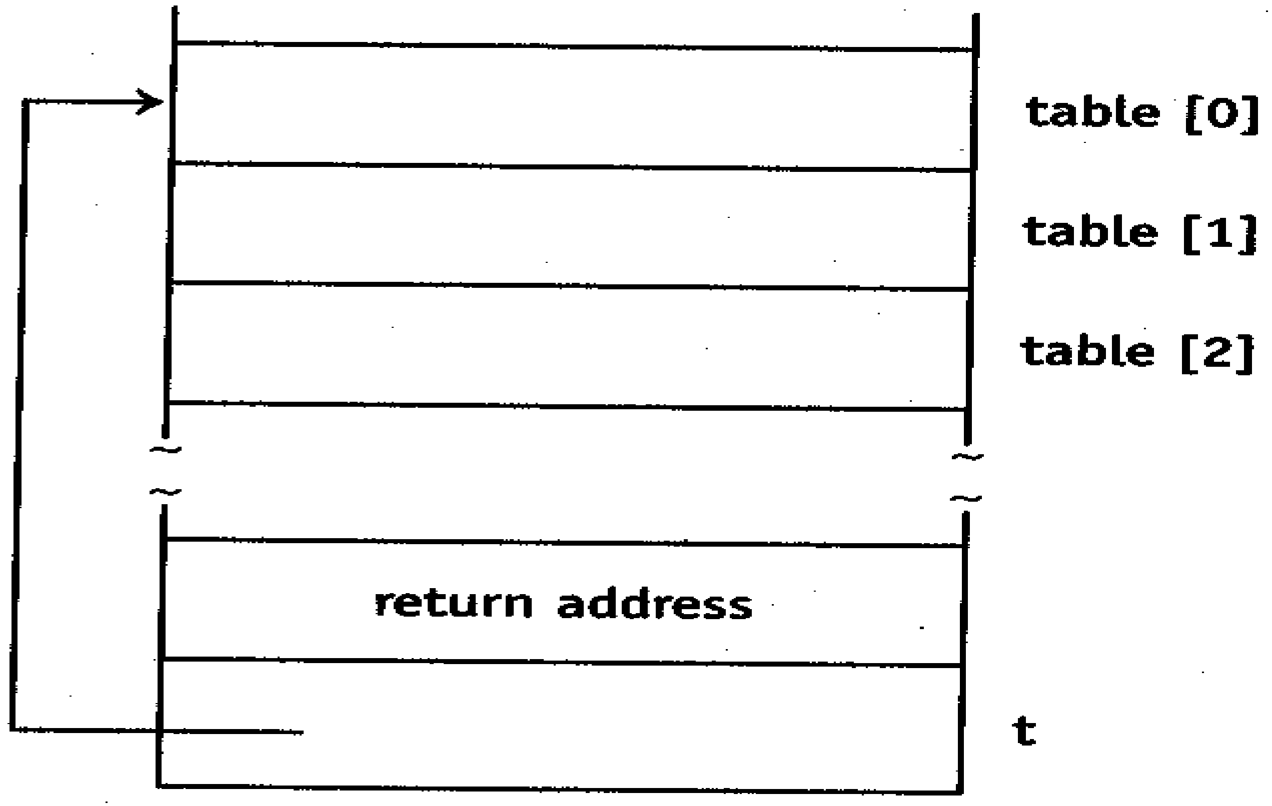
Arrays as arguments

FIGURE 8.24

```
int table[3];
void tabfun(int t[])   // t is parameter
{
    t[2] = 5;
}
void main()
{
    tabfun(table);     // array table is an argument
}
```

t is a pointer to table[0]. It can be used as a pointer, or as if it were the name of the array it is pointing to.

FIGURE 8.25



Equivalent forms

FIGURE 8.26 a)

```
void tabfun(int *t)
{
    *(t+2) = 5;          // use t as an int pointer
}
```

b)

```
void tabfun(int *t)
{
    t[2] = 5;            // use t as an array name
}
```

c)

```
void tabfun(int t[])
{
    *(t + 2) = 5;        // use t as an int pointer
}
```

FIGURE 8.27

```
@tabfun$pi:
                                     ;t[2] = 5;
          ldc   5                      ; get 5
          push                         ; prepare for sti
          ldc   2                      ; get index 2
          addr 2                       ; get address of t[2]
          sti                          ; store 5 in t[2]

          ret
;==============================================

main:     ldc   table                  ; call tabfun(table);
          push
          call @tabfun$pi
          dloc 1

          halt
table:    dw    3 dup 0
          end   main
```

Control statements

- while
- do-while
- for
- if
- if-else

Assembly code for while

```
while (x) {     // assume x is a global int
    cout << x << endl;
    x++;
}
```

where x is a global variable is

```
@L0:  ld x
      jz @L1          ; jump on false

      ld x            ; cout << x << endl;
      dout
      ldc '\n'
      aout

      ldc 1           ; x++
      add x
      st  x

      ja @L0          ; jump always
@L1:
```

while

FIGURE 8.28 C++ form

```
while (expression)
   s;
```

Assembly form

```
Evaluate expression

Jump on false to

Code for s

Jump always to
```

do-while

```
do {
   s;
} while (expression);
```

```
Code for s

Evaluate expression

Jump on true to
```

for

```
for (s1; expression; s2)
    s3;
```

```
Code for s1

Evaluate expression

Jump on false to

Code for s3

Code for s2

Jump always to
```

If-else

```
if (expression)
   s1;
else
   s2;
```

```
Evaluate expression

Jump on false to

Code for s1

Jump always to

Code for s2
```

if

```
if (expression)
   s;
```

```
Evaluate expression
Jump on false to

Code for s
```

Better translation of while

```
Jump always to

Code for s

Evaluate expression

Jump on true to
```

This form has the same size as the form in Figure 8.28; however, each iteration of the loop now requires the execution of only one jump instruction (the jump on true) instead of two. Thus, this form runs faster. Rewriting the previous **while** loop in this form, we get

```
      ja @L2            ; jump always

@L3:  ld   x            ; cout << x << endl;
      dout
      ldc  '\n'
      aout

      ldc  1            ; x++
      add  x
      st   x
@L2:  ld   x
      jnz  @L3          ; jump on true
```

Signed and unsigned comparisons

- Cannot be done easily on H1 because H1 does not have the S, V, and C flags.
- Subtracting two signed numbers and looking at the sign of the result yields the incorrect answer if overflow occurs.
- For now, we will assume overflow does not occur when performing comparisons.

Error in signed comparison

Compare 8000 and 0001 by subtracting and testing the sign of the result:

```
 8000  (-32,768)
 FFFF (-1)
 ---------
 7FFF
```

Positive result implies (incorrectly) that the top number is bigger than the bottom number.

Incorrect assembly code (wrong if overflow occurs)

FIGURE 8.29

C++	Assembly Language
`if (x>=y) {`	`ld x`
	`sub y`
	`jn @LO`
.	.
.	.
.	.
`}`	`@LO:`

Multi-word addition

Cannot be done easily on H1.

z = x + y;

where x, y, and z are double words is translated to

```
ld    x + 1
add   y + 1          ; what if carry out?
st    z + 1
ld    x
add   y
st    z
```

Bit-level operations

- Cannot be done easily on H1.
- Only jzop and jn are available on H1 for bit-level operations. These instructions test the msb in the ac register.

A program that converts decimal to binary. How to perform ‘&’ operation on line 13?

FIGURE 8.30

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 void main()
 5 {
 6   int x;
 7   int  count = 16;                  // count = number bits in int
 8   int mask = 0x8000;                // mask has only leftmost bit= 1
 9   cout << "Enter a decimal number: ";
10   cin >> x;
11   cout << "Binary equivalent = ";
12   for (int i = 1; i <= count; i++) {
13       if (x & mask)                 // bitwise AND; leftmost bit == 1?
14           cout << 1;
15       else
16           cout << 0;
17       x = x << 1;                   // left shift x one position
18   }
19   cout << endl;
20 }
```

A *recursive function* is a function that calls itself.

What is the output of the program on the next slide?

FIGURE 8.31

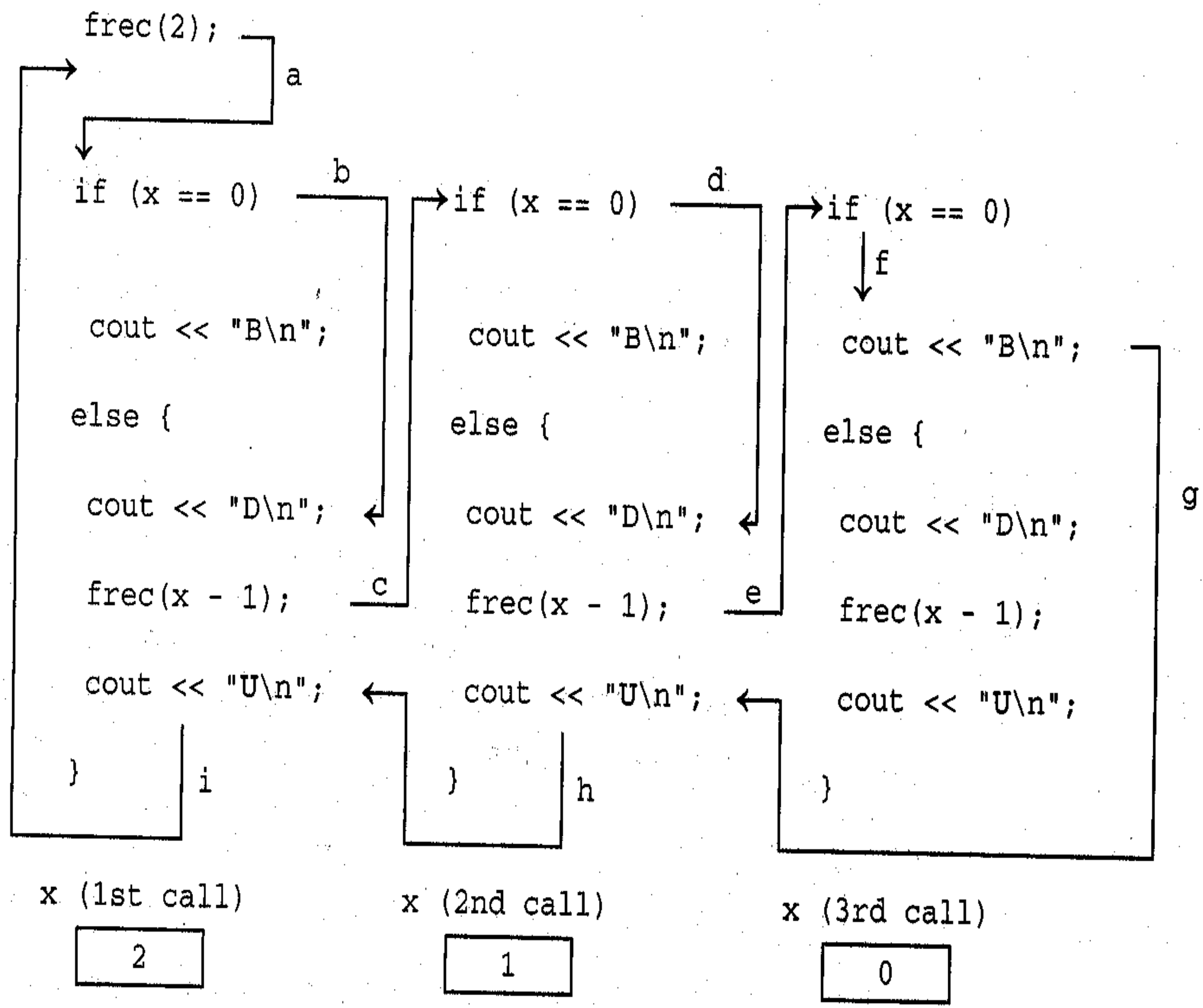
```
#include <iostream>
using namespace std;

void frec(int x)                        // recursive function
{
     if (x == 0)
          cout << "B\n";
     else {
          cout << "D\n";
          frec(x - 1);                  // recursive call
          cout << "U\n";
}
}
void main()
{
     frec(2);
}
```

Output of program on preceding line is

D
D
B
U
U

Statements preceding the recursive call are executed “on the way down”.
Statements following the recursive call are executed “on the way up”.

FIGURE 8.32


frec(2);
a
if (x == 0)
b
cout << "B\n";
else {
cout << "D\n";
frec(x - 1);
c
cout << "U\n";
}
i
if (x == 0)
d
cout << "B\n";
else {
cout << "D\n";
frec(x - 1);
e
cout << "U\n";
}
h
if (x == 0)
f
cout << "B\n";
else {
cout << "D\n";
frec(x - 1);
cout << "U\n";
}
g
x (1st call)
2
x (2nd call)
1
x (3rd call)
0

When the compiler translates a recursive function to assembly language, it does not treat a recursive function call in any special way. For example, the recursive call within the `frec` function

```
frec(x - 1);
```

is translated in the usual way:

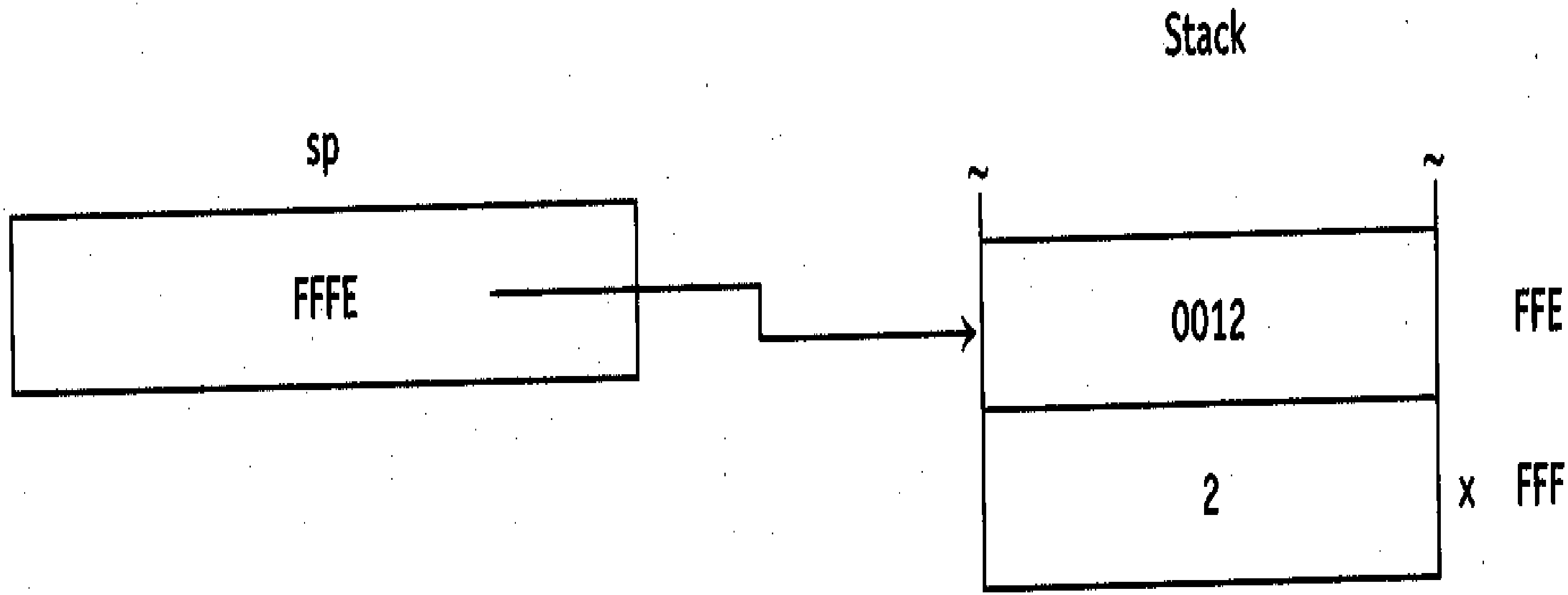
```
ldr 1           ; get value of x
sub @1          ; compute value of x - 1
push            ; create parameter on stack (the next x)
call @frec$i
dloc 1          ; remove the parameter from the stack
```

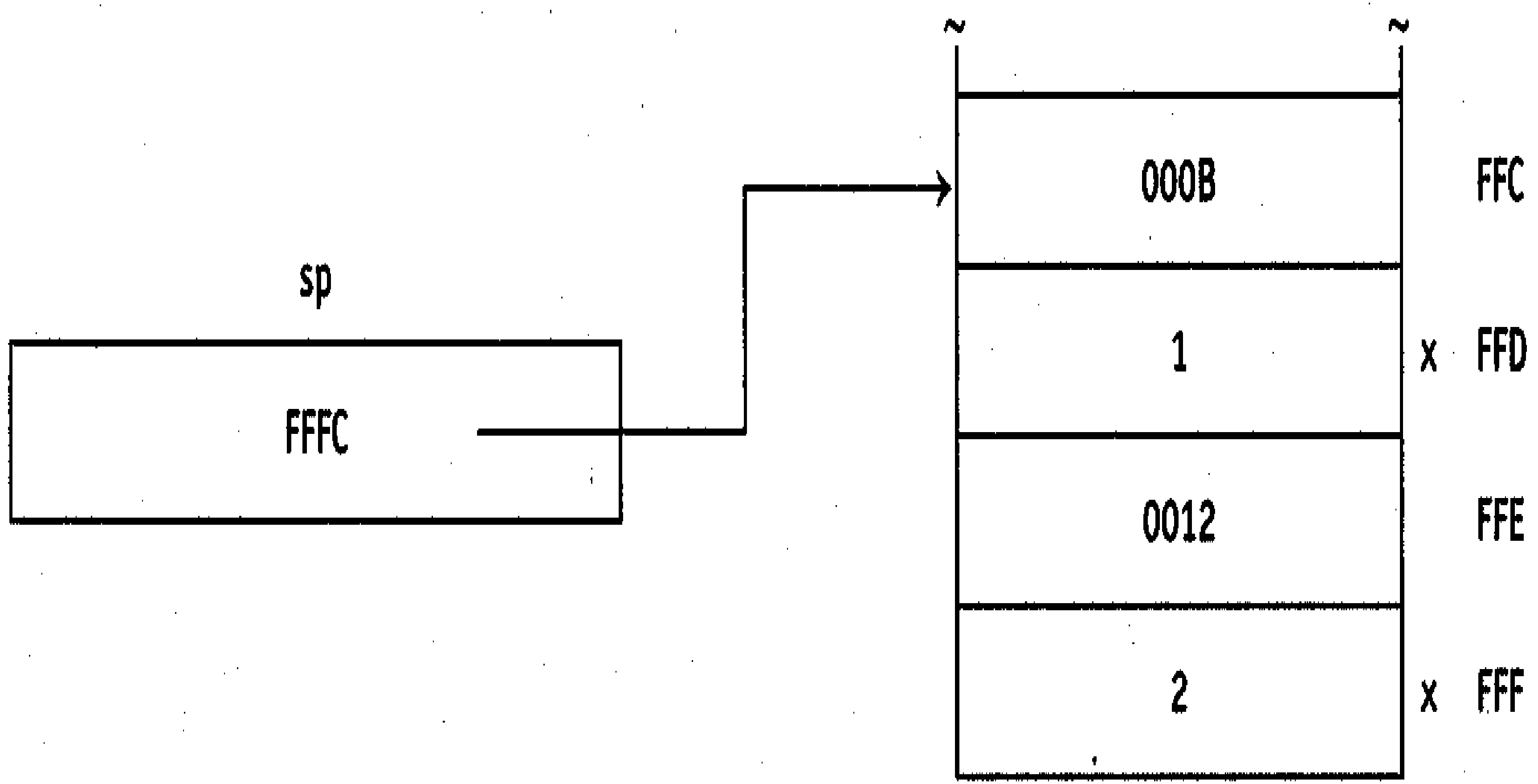
FIGURE 8.33

```
@frec$i:  ldr   1           ; if (x == 0)
          jnz   @L0

          ldc   @m0         ; cout << "b\n";
          sout

          ja    @L1

@L0:      ldc   @m1         ; cout << "D\n";
          sout

          ldr   1           ; frec(x - 1);
          sub   @1
          push
          call  @frec$i
          dloc  1

          ldc   @m2         ; cout << "U\n";
          sout

@L1:      ret
;=========================================
main:     ldc   2           ; frec(2);
          push
          call  @frec$i
          dloc  1

          halt
@m0:      dw    "B\n"
@m1:      dw    "D\n"
@m2:      dw    "U\n"
@1:       dw    1
          end   main
```

ldr 1

at the beginning of frec (line 1 of program on preceding slide) always loads the appropriate x.


Stack
sp
FFFE
0012
FFE
2
x
FFF


sp
FFFC
000B
1
0012
2
FFC
x
FFD
FFE
x
FFF

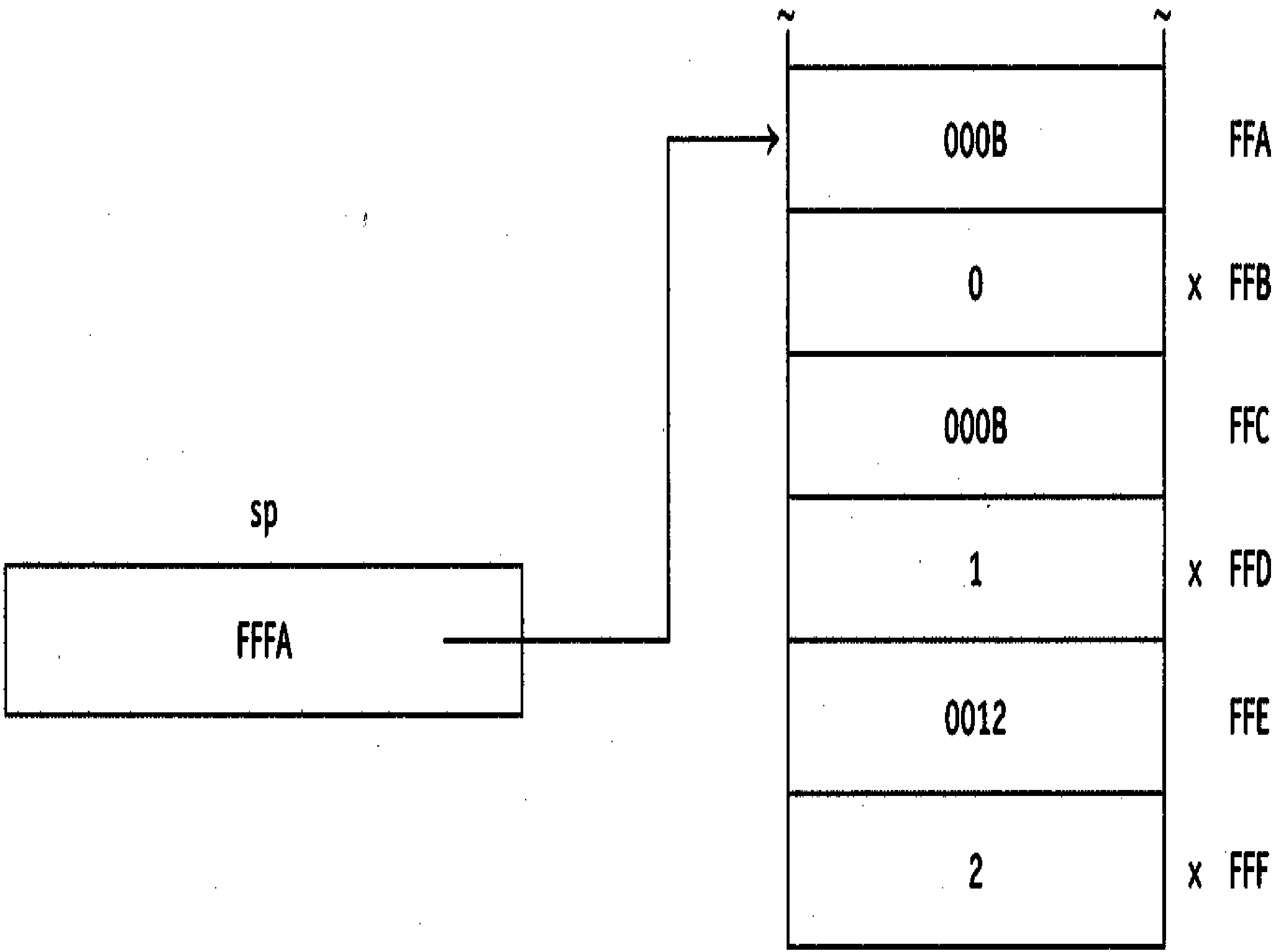

sp
FFFA
000B
0
000B
1
0012
2
FFA
x FFB
FFC
x FFD
FFE
x FFF

FIGURE 8.34

```
Starting session.  Enter h or ? for help.
---- [T7] F: ldc  /8 000/ b0
Machine-level breakpoint set at 0
---- [T7] F: ldc  /8 000/ n
No display mode
---- [T1] g
Machine-level breakpoint at 0
---- [T1] d$
FFE:  0012  0002                                first x
---- [T1] g
D
Machine-level breakpoint at 0
---- [T1] d$
FFC:  000B  0001  0012  0002                    first x
---- [T1] g
D
                                                second x
Machine-level breakpoint at 0000
---- [T1] d$
FFA:  000B  0000  000B  0001  0012  0002        first x
---- [T1] g
B
U                                               second x
U                         third x
Machine inst count =    23 (hex) =      35 (dec)
---- [T1] q
```